An Extension of PROLOG for Object-Oriented Programming in Logic

A. Schmidt, F. Belli
The University of Paderborn, Federal Republic of Germany

Abstract

In this paper, we attempt extending Logic Programming "smoothly" in order to allow object-orientation in a PROLOG-like environment. We call our extension "PROLoop" (Yet another *PRO*LOG-based *L*anguage for *O*bject-*O*riented *P*rogramming). PROLoop is the essential component of a PROLOG-based environment ("PROViro") to develop knowledge and rule-based expert systems.

PROViro consists of a series of pragmatic components as to testing (PROTest), knowledge version control (PROVers), self actualization of the documentation (PROSelf), etc. The potential of PROLoop stems from its simplicity. This simplicity makes PROLoop easy to use and to extend, allows to achieve a high degree of reliability of PROLoop programs, increases their maintainability, etc. Because of its artlessness, PROLoop is also a good example for understanding and teaching object-oriented programming. Nevertheless, PROLoop possesses sufficient expression power which we demonstrate by including non-trivial examples produced in a real project.

VISIT…

1. Introduction

Object-oriented Programming is increasingly becoming the role of the structured programming when developing large software systems. On the other hand, Logic Programming represents a natural manner to implement declarative knowledge efficiently, e.g. to develop expert systems. Therefore, coupling these two programming paradigms becomes a real challenge when one has to implement knowledge-based systems in a modern, comfortable environment.

Over the past few years it has become popular to design object-oriented programming languages, often as add-ons to LISP (e.g. LOOPS, Flavors etc.) and in a few cases to PROLOG (e.g. SPOOL [12]). This paper reviews a pragmatic approach to adopt concepts of the object-orientation in PROLOG. Instead of the trivial way through achieving an interface from an object-oriented language as Smalltalk to PROLOG, or v.v., we take the "tough" way: Extending PROLOG smoothly allowing object-orientation in a PROLOG-like environment. We call our extension PROLoop: A Yet Another PROLOG-based Language for Object-Oriented Programming. PROLoop is the essential component of our environment for PROLOG (PROViro) which includes also a test environment (PROTest), knowledge version control (PROVers), documentation and its self-actualization (PROSelf), etc.

The major aspect why we set our sights on the design of PROLoop is its simplicity. As an example of this simplicity, we have not included a multiple inheritance mechanism as a built-in feature which would increase the complexity of the basic system. This could however be implemented by the user through an appropriate extension, when necessity arises. PROLoop programs are easy to be read and understood, and therefore, they increase maintainability. As a pleasant side-effect, PROLoop is - because of its artlessness - a good example for understanding and teaching object-oriented programming.

In the following chapters we will survey PROLoop, also demonstrating its power to express real project applications in a simple manner. To exclude likely misunderstandings, we review the basic concepts of the Object-Oriented Programming next.

2. Elements of Object-oriented Programming Paradigm

Objects: Classes and Instances

Most (but not all) object-oriented programming systems have *classes* and *instances* as their major elements. One may casually compare classes with types in some procedural programming languages like Pascal, Ada etc. In a hierarchical structure, classes contain description of similar objects. Every class has a fatherclass which represents its *superclass*. Objects of a class possess the properties of its superclass. The only class without superclass is called *root class*. Instances are "visible", i.e. concrete examples of a class. As an example, the class PC Brand xxx may have Fred's PC Brand xxx and Barney's PC Brand xxx as instances. Objects provide *data encapsulation* (*information hiding*) as a technique for minimizing interdependencies among separately implemented modules and data abstraction as a form of modular programming (see also [1-7]). Data and operations on this data are called the *inner state* of an object.

Class and instance variables

Class variables are variables whose values are shared by all instances of the class. Such variables are accessible only in classes. This contrasts with instance variables which provide local storage in instances. Sometimes instance variables are called *slots*.

Methods

Methods are local functions operating on the inner state of an object. In terms of such functions, the response of an object will be determined when a message arrives at this object. Some basic methods of the root-class are creating a new instance, reading and writing instance variables, i.e. changing the inner state of an object, etc.

Inheritance

A class inherits variables and methods from its superclasses. A simple kind of inheritance is *hierarchical inheritance* where a class is defined in terms of a single superclass. *Multiple inheritance* means that a class may have more than one single superclass. This increases the sharing potential of the descriptions of existing classes. It is obvious that the potential of code sharing is greater, but the likeliness of conflicts between multiple superclasses and their handling increase the complexity of systems providing multiple inheritance.

Messages

The communication between objects will be materialized through sending messages. An incoming message causes the receiver to prompt a reaction which is similar to a procedure call. A message can provoke different responses when arriving at various objects. The set of all messages available for an object is called a *protocol.*

Message passing materializes the communication between objects.

3. "Match-making" PROLOG and Object-Orientation

The merits of object-oriented programming, especially data encapsulation and inheritance as mentioned above, support structured design and enhance maintainability of large software systems in a natural way. However object-orientation lacks built-in functions which are contained in most knowledge-based applications, e.g. generalized facilities for deductive retrieval and pattern matching [8]). On the other hand, PROLOG has built-in deductive retrieval through backtracking and pattern matching via unification. Thus, coupling both paradigms (i.e. object-oriented and logic programming) allows to use their advantages to the most benefit. To enable such a "marriage" of these paradigms, we could alternatively choose from following:

- implementing object-orientation as an add-on to PROLOG,
- implementing a PROLOG-like deductive retrieval mechanism in an object-oriented environment [9] and
- building a bridge between two existing systems based on object-oriented and logic programming [8].

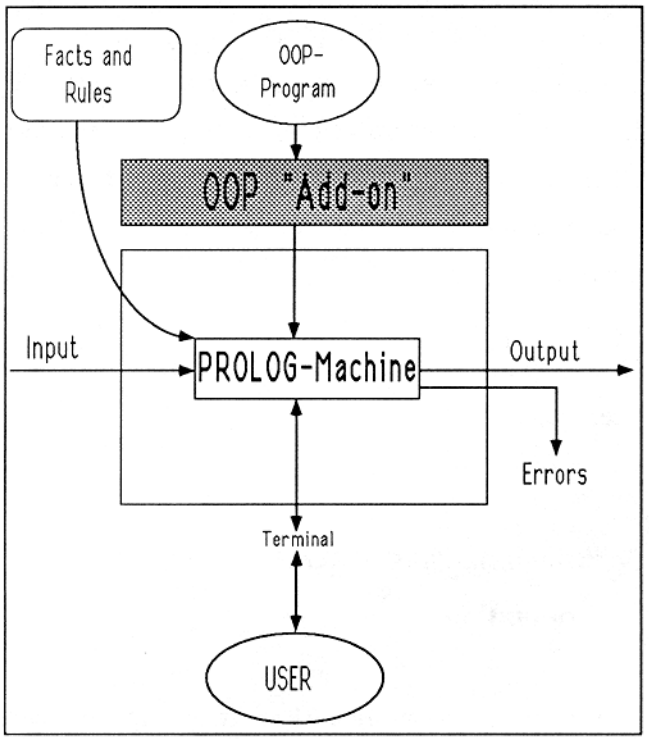

Fig. 1. Object-orientation as an add-on to PROLOG.

In this paper, we investigate the first possibility, i.e. implementing object-orientation as an add-on to PROLOG (Fig. 1). For this purpose, objects will be represented as predicates of a PROLOG program. Class and instance variables are treated as arguments. This also provides a natural materialization of the information hiding principle. A number of clauses to a common functor materialize a PROLOG data capsule (Fig. 2).

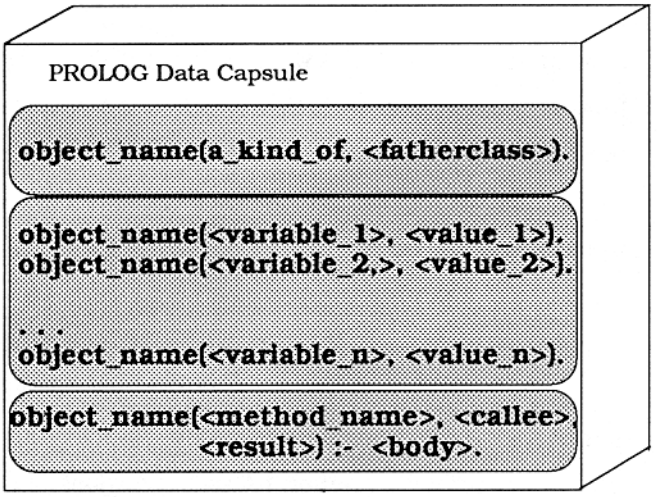

Fig. 2. PROLOG data capsule

Methods are realized as the body of a rule; the arguments in the head of the rule constitute selectors in order to choose the adequate method.

Message passing in PROLoop:

Messages in PROLoop are twofold:

- *local* messages and
- *global* messages.

Local messages can be executed only by the object itself, i.e. they are not *visible* outside of the object. Global messages are generally accessible, i.e. they can be executed by any object. Accordingly, the user of PROLoop has to specify a method to be either "local" or "global".

As an example, next we construct an instance of the class computer_Brand_X in PROLOG to demonstrate the idea depicted in Fig. 2. The example in Fig. 3 shows that the name of the instance operates as a functor.

```
my_computer(a_kind_of, computer_Brand_X).
my_computer(cpu, intel_80286).
my_computer(memory, '640_KB').
my_computer(keyboard, keys_80).
my_computer(monitor, monochrom_ega).
my_computer(storage_device, floppy_5_14).
```

Fig. 3. Example for the representation of an instance in the PROLOG knowledge base

Classes are represented through the class predicate:

```
class(<class>, <fatherclass>, <list of variables>).
```

A class can be created with the statement defclass and methods with the statement defmethod:

```
:- defclass(<class>,<fatherclass>,<list of variables>).
:- defmethod(<class>,<methodname>,<local or global?>,
             <receiver>,<result>,<method>).
```

Following, we realize methods in PROLOG code adopting an object-oriented syntax. As an example, we define a method to calculate the price of a specific computer:

```
computer_brand_X(price, local, SELF, RESULT) :-
   /* receiver is the object itself */
   /* Smalltalk-like syntax:
      get the cpu type of a given computer and
      store the result in the PROLOG variable "CPU" */
   CPU <- SELF : get(cpu),
   /* PROLOG syntax:
      calculate the price for three different
   CPU types */
   (( CPU == 8088,  RESULT = 1000, ! ) ;
    ( CPU == 80286, RESULT = 2000, ! ) ;
    ( CPU == 80386, RESULT = 4000, ! )).
```

Fig. 4. Example for a method to demonstrate the idea depicted in Fig. 2.

Further, we combine all OR-branches to one clause body. Incidentically the necessity can emerge to refer to an external method definition as a procedure call, e.g. to invoke a recursion as the following example demonstrates.

```
object(<methodname>, local, SELF, RESULT) :-
/* suffix "0" in methodname as a transition argument */
   object(<methodname>0, SELF, RESULT).

object(<methodname>0, SELF, RESULT) :-
   /* next level: suffix "1" ... */
   object(<methodname>1, SELF, RESULT, [ ], LIST).
   /* termination */
object(<methodname>1, SELF, RESULT, RESULT, [ ]).
object(<methodname>1, SELF, RESULT, LIST, [ H | T ]) :-
/* recursive procedure call */
   object(<methodname>1, SELF, RESULT, [ H | LIST ], T).
```

Fig. 5. External method definition as a procedure call

4. Examples and Comparison with Related Work

In this chapter we will explain some relevant facilities of PROLoop by means of two examples. The first example has been kept very primitive in order to assure easy understanding through minimizing the amount of the additional non-PROLOG syntax which we introduced to obtain the object orientation. We will then apply this example in an interactive environment to demonstrate its flexible employment. The second example handles a more complex problem: Transport of dangerous goods.

Example 1. *A simple Object-Oriented Program.*

As an introductory example, Fig. 6 represents a trivial PROLoop program.

```
/* Define classes: class name, superclass, class
   variables */
:- defclass(car, object, [ price := 0 ])
      /* Default value for price */,
      defclass(mercedes190, car, [ ]),
      defclass(porsche911, car, [ ]).
```

Fig. 6. Introductory example for PROLoop

Following, in Fig. 7 we give a sample session to interact the program given in Fig. 6.

```
/* Create a new instance; "new" is a built-in method */

?- my_car <- mercedes190 : new.

/* Change initial value for price with built-in
   method put */

?- my_car : put(price := 25000).

/* Listing of object my_car */

?- listing(my_car).

/* PROLOG output: */

   my_car(a_kind_of, mercedes190).
   my_car(price, 25000).
```

Fig. 7. A sample session to the PROLoop program of Fig. 3

We already mentioned that some other authors have also attempted to introduce object orientation into logic programming, e.g. ALF, SPOOL, etc. [9, 12]. As we studied these publications carefully, we understood that SPOOL has similar design goals as PROLoop, e.g. simplicity, transparency of its constructs, etc. We are, however, still convinced that PROLoop has some additional qualities which make it worthwhile to be studied. To demonstrate these qualities, we will convert the above introductory example into SPOOL which will be represented in Fig. 8.

```
/* Creating classes */

class car has
   super-class root-class;
   instance-var price;
   /* methods are realized for example here */
   methods
      <rule_or_fact 1>; ... ;
      <rule_or_fact n>;
end.

class mercedes190 has
   super-class car;
end.

class porsche911 has
   super-class car;
end.

/* Instances */
instance my_car is-a mercedes190 where price : 25000.
```

Fig. 8. The introductory program of Fig. 6 in SPOOL

The program depicted in Fig. 8 reveals the major weakness of SPOOL which stems from its expression power: When designing SPOOL, the designers' intention was to impressively demonstrate the elegance and potential of the combination of logic and object-oriented programming [12]. Therefore, SPOOL programmers must be alerted: The interpreter realizes changes on the value of instance variables as side-effects which affect the PROLOG knowledge base. Thus, changes have not been recovered when backtracking occurs.

Example 2. *Transport of Dangerous Goods.*

The code given in Fig. 9 has been excerpted from the program system of FIREX (FIRe avoidance and Combat EXpertise), a knowledge-based system for the Transport of Dangerous Goods and Fire Departement Consulting [10]. Maritime regulations define different rules to segregate substances while transporting them (e.g. away from, longitudinally separated from etc.). For the shipment of cargos, nine classes have been introduced as a guideline to consider properties of dangerous substances. The classes are further divided into subclasses (Please do not mix these *classes of substances* which express the transport risk with the *classes of object-oriented paradigm*. We will synonymously use the phrase *danger categories* instead of *danger classes*). The table containing segregation distances is published in official guidelines for the storage and stowage of dangerous goods [11]. In addition to this table, there exists a great number of exceptions to the regular segregation rules. The following examples depict simplified programs of the mentioned system FIREX materializing the introduced raw expertise: We will define *danger categories* and *substances* as classes in the sense of object-orientation. A method of the class substance is implemented to determine the segregation rules when substances are given. Classes with substances as subclasses have the segregation rules as instance variables. These rules correspond to the class concerned and its exceptions.

The PROLoop program essentially consists of two parts: *Class definitions* and *method definitions*. Through class defintions the programmer maps the class hierarchy and declares class variables and assigns default values. Danger categories like "explosive", "inflammable solid", "poison", "corrosive", etc. and also substances which belong to these danger classes are declared first. Please note that some danger classes may have subclasses, e.g. danger class 3 ("inflammable liquids") has three subclasses: 3.1 ("low inflammation"), 3.2 ("middle inflammation") and 3.3 ("high inflammation").

```
/* Definition of some sample classes
   ("object" is always root class) */

:- defclass(substance, object, [ ]).

/* "4" and "x" are abbreviated segregation distances */

:- defclass('class 5', substance, [ ]),
   defclass('class 5.1', 'class 5',
            [[ segregation_table := [[ 'class 1.1', 4 ],
                                      ...
                                    [ 'class 9', x ]]]).
:- defclass('Ammoniumnitrate', 'class 5.1',
            [exceptions := [[ 'Calciumpermanganate', 2],
                           [ 'Caliumpermanganate', 2 ],
                           [ 'Epichlorohydrin', 2 ]].


/* Now we define a method to determine the segregation
   distance when two substances are given. "is_instance"
   and "in_list" are user-defined PROLOG-rules.
   "subst_name" and "dangerclass" are user-defined
   methods and "get" is a built-in method of "object". */

:defmethod(substance,global,segregation(S),SELF,RESULT,
       /* method algorithm */
       (( is_instance(SELF,_),
          is_instance(S,_),
          EXCEPTLIST <- SELF : get(exceptions),

          /* find out regular segregation distance */
          (( SUBSTANCE <- S : subst_name,
             not (in_list(SUBSTANCE, EXCEPTLIST, _),
             TABLE <- SELF : get(imdg_table),
             CLASS2 <- S : dangerclass,
             in_list(CLASS2,TABLE,RESULT), ! ) ;

          /* exception, no regular segregation
             distance */
           ( not (EXCEPTLIST == [ ]),
             SUBSTANCE <- S : subst_name,
             in_list(SUBSTANCE,EXCEPTLIST,RESULT),
             ! ))) ;

       /* Error, return to top level of PROLOG */
       ( not(is_instance(SELF,_) ; is_instance(S,_)),
         write('Message is not allowed for classes!'),
         abort ) ;

       /* Never fail ... */
       true )).
```

Fig. 9. Code fragment for the transport of dangerous goods, implemented in PROLoop

```
/* Sample Session: first creating instances;
   consider 3 boxes to store */

?- box1 <- 'Ammoniumnitrate'  : new.
?- box2 <- 'Ammoniumnitrate'  : new.
?- box3 <- 'Epichlorohydrin'  : new.

/* Reading instance variables ... */

?- R <- box1 : get(exceptions).
R = ['Calciumpermanganate',2], ['Caliumpermanganate',2],
    ['Epichlorohydrin',2]]

/* Ask for segregation distances */

?- SD <- box1 : segregation(box2).

/* both boxes contain the same substance; "x":
   no segregation necessary */

SD = x
?- SD <- box1 : segregation(box3).

/* Exception: instead of segregation distance "1"
   ("away from")
   for dangerclasses "5.1" and "6.1" (segregation
   table): "2" ("separated from") */

SD = 2
```

Fig. 10. Sample session of the program represented in Fig. 9.

Methods are materialized through PROLOG and object-oriented code (see the following example and view the comments). We implemented as an example one single segregation rule as a method with the class substance as owner.

5. Conclusion

Object-oriented programming is a natural manner to implement complex programming tasks, especially when hierachically organized data objects are concerned. Logic programming provides built-in facilities for backtracking and unification which are indispensable for applications requiring deductive retrieval. A marriage of object-oriented paradigm with logic programming supply advantages of both paradigms. For this purpose, we develop PROLoop which is embedded in a sophisticated PROLOG environment (PROViro). PROViro provides a series of components as to document and test expert systems, to control different versions of represented knowledge, etc. PROLoop is easy to use and to extend. This ensures a high degree of the reliability and maintainability of PROLoop programs through reduced complexity.

The addresses of the authors:

F. Belli, A. Schmidt,

The University of Paderborn, Dept. of Electrical and Electronics Engineering, Pohlweg 47-49, D-4790 Paderborn, Federal Republic of Germany

phone: (+49)5251-60-3819

fax: (+49)5251-60-2519

telex: 936776 unipb d

e-mail: belli%adt-alpha@pbinfo

References

[1] Pascoe, G.A., Elements of Object-Oriented Programming, McGraw-Hill BYTE-Magazine August 1986, pp. 15-20

[2] Rentsch, T., Object Oriented Programming, ACM SIGPLAN Notices, Vol. 17, No. 9, Sept. 1982, pp. 51-57

[3] Snyder, A., Encapsulation and Inheritance in Object-Oriented Programming Languages, Proc. OOPSLA '86, Sept. 1986, pp. 38-45

[4] Stefik, M. and Bobrow, D.G., Object-Oriented Programming: Themes and Variations, AI Magazine, Vol. 6, No. 4, Winter 1986, pp. 40-62

[5] Thomas, D., What's in an Object, McGraw-Hill BYTE Magazine, Vol. 14, No. 3, March 1989, pp. 231-240

[6] Nygaard, K., Basic Concepts in Object Oriented Programming, ACM SIGPLAN Notices, Vol. 21, No. 10, Oct. 1986, pp. 128-132

[7] Robson, D., Object-Oriented Software Systems, McGraw-Hill BYTE Magazine, Vol. 6, No. 8, Aug. 1981, pp. 74-86

[8] Koschmann, T., Evens, M.W., Bridging the Gap between Object- Oriented and Logic Programming, IEEE Software 7/88: "Shedding Light on 4GL's", pp. 36-42

[9] Mellender, F., An Integration of Logic and Object-Oriented Programming, ACM SIGPLAN Notices, Vol. 23, No. 10, pp. 181-185

[10] Belli, F. et. al., Some Aspects on the Developement and Validation of FIREX - a Knowledge-Based System for the Transport of Dangerous Goods and Fire Department Consulting, Proceedings 2nd Intern. Conference of IEA/AIE - 89, June 6-9, 1989, Vol. II, pp. 680-689

[11] Stowage & Segregation Guide to IMDG-Code, K.O. Storck Verlag, Hamburg.

[12] Fukunaga, K. und Hirose, S., An Experience with a PROLOG-based Object-Oriented Language, Proc. OOPSLA '86, pp. 224-231